中文翻译链接

<https://weread.qq.com/web/reader/751326d0720befab7514782k32b321d024832bb90e89958>

**翻译时修饰语提前**

*献辞*

*To Jordan, for her* ***unwavering*** *support no matter*

*how many times she heard “it’s almost done.”*

MATT FRISBIE 马特弗里斯比

ABOUT THE AUTHOR

MATT FRISBIE has worked in web development for over a decade. During that time, he’s been a

**startup** co-founder, an engineer at a Big Four tech company(四大科技公司之一), and the first engineer at a Y **Combinator** startup that would **eventually** become a **billion-dollar** business. As a Google software engineer, Matt worked on both the **AdSense** and **Accelerated** Mobile Pages (AMP) platforms; his code contributions run on most of the planet’s web browsing devices.

**Prior to this,** Matt was the first engineer at **DoorDash**, where he **laid the foundation** for their driver scheduling, menu management, and order **dispatch infrastructure**. Matt has written two books and recorded two video series for **O’Reilly and Packt**, speaks at **frontend meetups and webcasts**, and is a Level 1 sommelier. He majored in computer engineering(计算机科学) at the University of Illinois(伊利诺斯州（美国州名）) at Urbana–Champaign(伊利诺伊大学香槟分校). Matt’s Twitter handle is @mattfriz.

目录

CHAPTER 10: FUNCTIONS 349

1. Arrow Functions 350
2. Function Names 351
3. Understanding Arguments 353

Arguments in Arrow Functions 355

1. No Overloading 355 **（pdf P398）**
2. Default Parameter Values 356

Default Parameter Scope and Temporal Dead Zone 358

1. Spread Arguments and Rest Parameters 359

Spread Arguments 359

Rest Parameter 360

1. Function Declarations versus Function Expressions 361

Functions as Values 362

1. Function Internals 364

arguments 364

this 365

caller 366

new.target 367

1. Function Properties and Methods 367
2. Function Expressions 370
3. Recursion 372
4. Tail Call Optimization 373
5. Tail Call Optimization Requirements 374

Coding for Tail Call Optimization 375

Contents

xxvi

1. Closures 376

The this Object 379

Memory **Leaks** 381

1. Immediately Invoked Function Expressions 382
2. Private Variables 384

Static Private Variables 385

The Module Pattern 387

The Module-Augmentation Pattern 388

1. Summary

目录注释

Leak /li:ks/泄漏 英/liːks/

**CHAPTER 6 Collection Reference Types**

Collection ：收藏品，物；

6.1 THE OBJECT TYPE

Although instances of Object don’t have much **functionality**(功能), they are **ideally** suited to **storing and transmitting data** around an application(应用程序).

There are two ways to explicitly create an instance of Object:

* The first is to use the new operator with the Object constructor
* The other way is to use object literal notation.

In this example, the left **curly brace** ({) **signifies** the beginning of an object literal because it occurs in an expression context(表达式上下文).

**CHAPTER 8 Objects, Classes, and Object-****Oriented Programming**

it helps to think of ECMAScript objects as hash tables(哈希表): nothing more than a grouping of namevalue pairs where the value may be data or a function

8.1 UNDERSTANDING OBJECTS

The **canonical**([典型](javascript:;)) way of creating a custom object is to create a new instance of Object and add properties and methods to it, as in this example:

主语 谓语 宾语

Code Example

The sayName() method displays the value of this.name, which resolves to person.name.

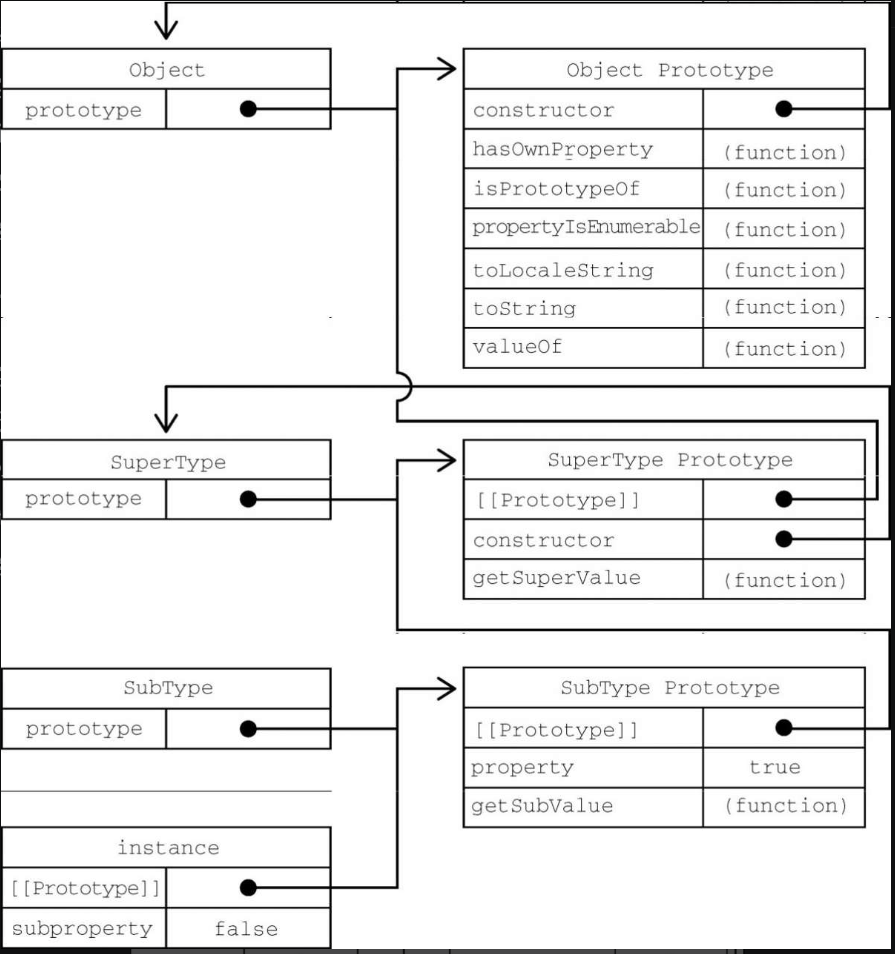
8.3 继承

总述：

javascript中的面向对象编程有关继承的实现方式只有“实现继承”，这种继承能继承实际的方法。实现继承主要通过原型链实现。原型链的基本构想是在实例和原型之间构造了一条原型链，

8.3.1 原型链

1．默认原型与图8-5展示了完整的原型链



2．原型与继承关系：确定原型与实例的关系的两种方式

原型搜索机制:

文脉：叙述过程，思考逻辑

3．关于方法：子类有时候需要覆盖父类的方法

原型链的问题

主要问题：原型包含引用值导致的继承问题。

属性通常会在构造函数中定义而不会定义在原型上的原因

“盗用构造函数”

function SuperType() {

this.colors = ["red", "blue", "green"];

}

function SubType() {

//继承SuperType

SuperType.call(this);

}

let instance1 = new SubType();

instance1.colors.push("black");

console.log(instance1.colors); // "red, blue, green, black"

let instance2 = new SubType();

console.log(instance2.colors); // "red, blue, green"

8.3.2 盗用构造函数

为什么要使用闭包，闭包有什么好处

**原型 作用域，作用域链解析，原型链，执行上下文，闭包**

4.2 执行上下文与作用域

变量或函数的上下文决定了它们可以访问哪些数据，以及它们的行为。

上下文中定义的所有变量和函数都在关联的变量对象（variable object）上,变量对象无法通过代码访问，但后台处理数据会用到它。

代 码

**第10章 函数**

本章内容

函数实际上是对象，定义函数有以下**四种**方式

* 函数声明
* 函数表达式
* 箭头函数（**函数表达式的简写版本**）
* 使用Function构造函数（**有性能问题**）

let sum = new Function("num1", "num2", "return num1 + num2"); // 不推荐

**10.1 箭头函数**

1. 箭头函数的定义：ECMAScript 6新增了使用胖箭头（=>）语法定义函数表达式的能力

(New in ECMAScript 6 is the capability to define a function expression using the fat-arrow syntax.

)

2.箭头函数的简洁性与简洁性带来的问题

**箭头函数是函数表达式的一种简写/省略版本：** 首先从左到右是可省略function关键字，省略括号，省略参数；省略大括号、return。

各自的省略情况如下：

* 括号：是否省略有参数个数决定：只有一个参数，可以省略，没有参数或有多个参数，不可省略；
* 大括号与return: 有return时{}不可省略,有{}时return可以没有

**{}的省略待完善**

**3.箭头函数中的参数**

虽然箭头函数中没有arguments对象，但可以在包装函数中把它提供给箭头函数：

注意 ECMAScript中的所有参数都按值传递的。**不可能按引用传递参数。如果把对象作为参数传递，那么传递的值就是这个对象的引用。（**也就是说不可能通过传递参数改变对象**）**

1. [14.2](https://262.ecma-international.org/6.0/" \l "sec-arrow-function-definitions) Arrow Function Definitions
   1. [14.2.1](https://262.ecma-international.org/6.0/" \l "sec-arrow-function-definitions-static-semantics-early-errors) Static Semantics: Early Errors
   2. [14.2.14](https://262.ecma-international.org/6.0/" \l "sec-arrow-function-definitions-runtime-semantics-iteratorbindinginitialization) Runtime Semantics: IteratorBindingInitialization

NOTE Arrow functions never have an arguments objects.

【https://262.ecma-international.org/6.0/#sec-functiondeclarationinstantiation】

**Arrow function expressions**

An **arrow function expression** is a compact alternative to a traditional [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), with some semantic differences and deliberate limitations in usage:

* Arrow functions don't have their own bindings to [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), or [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), and should not be used as [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method).
* Arrow functions cannot be used as [constructors](https://developer.mozilla.org/en-US/docs/Glossary/Constructor). Calling them with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) throws a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError). They also don't have access to the [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keyword.
* Arrow functions cannot use [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) within their body and cannot be created as generator functions.

箭头函数表达式的语法比[函数表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/function)更简洁，并且没有自己的[this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this)，[arguments](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments)，[super](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/super)或[new.target](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new.target)。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

**10.2 函数名**

Because function names are **simply(无非) pointers** to functions, they **act like** any other variable containing a pointer to an object. This means it’s possible to have multiple names for a single function, as in this example:

一个函数可以有多个名称。ECMAScript 6的所有函数对象都会暴露一个只读的name属性，其中包含关于函数的信息

**求函数名有以下几种情况：**

01.

函数声明的函数名

函数表达式的函数名

箭头函数的函数名（赋值与没赋值两种情况）

使用构造函数创建的函数名

02.

如果函数是一个获取函数、设置函数，或者使用bind()实例化

arguments对象是一个类数组对象

function sayHi() {

console.log('Hi,', arguments[0])

}

sayHi('xiaoxiang') // Hi, xiaoxiang

**10.3 理解参数**

Todo:格式调整

ECMAScript函数的参数在内部表现为一个数组

函数重载与函数签名（Signature (functions)）

[Signature (functions)](https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function)

A **function signature** (or type signature, or method signature) defines input and output of [functions](https://developer.mozilla.org/en-US/docs/Glossary/Function) or [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method).

A signature can include:

* [parameters](https://developer.mozilla.org/en-US/docs/Glossary/Parameter) and their [types](https://developer.mozilla.org/en-US/docs/Glossary/Type)
* a return value and type
* [exceptions](https://developer.mozilla.org/en-US/docs/Glossary/Exception) that might be thrown or passed back
* information about the availability of the method in an [object-oriented](https://developer.mozilla.org/en-US/docs/Glossary/OOP) program (such as the keywords public, static, or prototype).

**10.4 没有重载**

为什么ECMAScript没有函数重载：ECMAScript函数没有签名，因为参数是由包含零个或多个值的数组表示的。没有函数签名，自然也就没有重载。

函数声明与函数表达式的角度理解，把函数名当成指针也有助于理解

**10.5 默认参数值**

01.给参数传undefined时，实现默认参数的一种常用方式

* 隐式定义默认参数（在ECMAScript5.1及以前）：
* 显式定义默认参数（ECMAScript 6之后）：

02.arguments对象：

arguments对象，它始终以调用函数时传入的值为准，不受默认参数定义是值的影响，当函数内部修改默认参数的值时，arguments对象也不会改变。

03.求值默认参数：

是函数时：

04. 默认参数作用域与暂时性死区

给多个参数定义默认值实际上跟使用let关键字顺序声明变量一样。来看下面的例子：

function makeKing(name = 'Henry', numerals = 'VIII') {

return `King ${name} ${numerals}`;

}

console.log(makeKing()); // King Henry VIII

这里的默认参数会按照定义它们的顺序依次被初始化，所以后定义默认值的参数可以引用先定义的参数，但前面定义的参数不能引用后面定义的（参数初始化顺序遵循“暂时性死区”规则）。参数也存在于自己的作用域中，它们不能引用函数体的作用域。

05.参数初始化顺序遵循“暂时性死区”规则

**10.14 闭包**

标识符查找

var color = 'blue';

function getColor() {

return color;

}

console.log(getColor()); // 'blue'

[JS闭包直观上的说法就是:在存在函数嵌套的代码里，**内层**函数引用了**外层**函数的变量。](https://zhuanlan.zhihu.com/p/25296587)

理解作用域链创建和使用的细节

作用域链的概念：作用域链其实是一个包含指针的列表，每个指针分别指向一个变量对象，但物理上并不会包含相应的对象。

执行上下文的概念

活动对象的概念

在调用一个函数时，会为这个函数调用创建一个执行上下文，并创建一个作用域链。然后用arguments和其他命名参数来初始化这个函数的活动对象。

**函数的定义与调用**

代码

1. 在定义compare()函数时：

为它创建作用域链(保存在内部的[[Scope]])，预装载全局变量对象

compare {

[[Scope]]

}

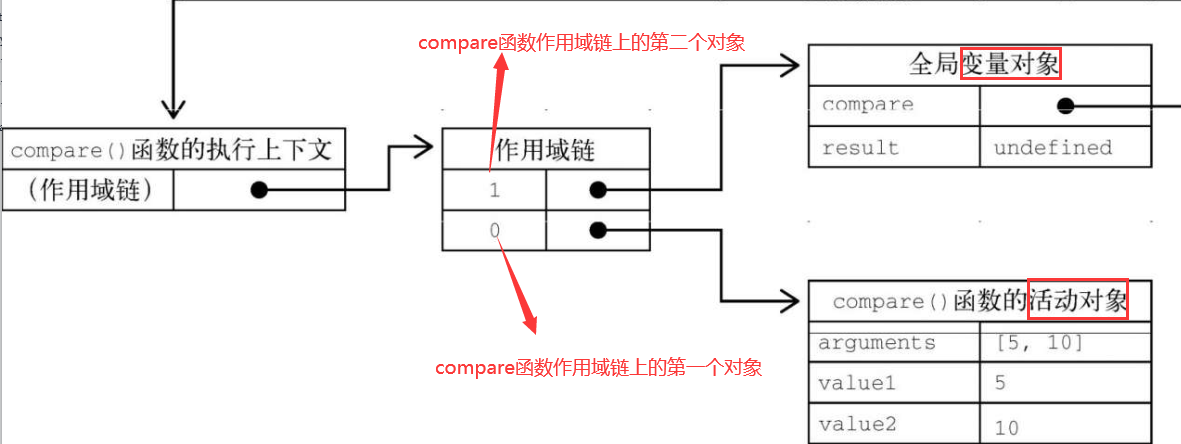
1. 调用时：(1)会创建相应的**执行上下文**，然后通过复制函数的[[Scope]]来创建其作用域链(相当于全局上下文的变量对象先装)。
2. 接着会创建函数的活动对象（用作变量对象）并将其推入作用域链的前端。

在函数执行时，要从作用域链中查找变量，以便读、写值

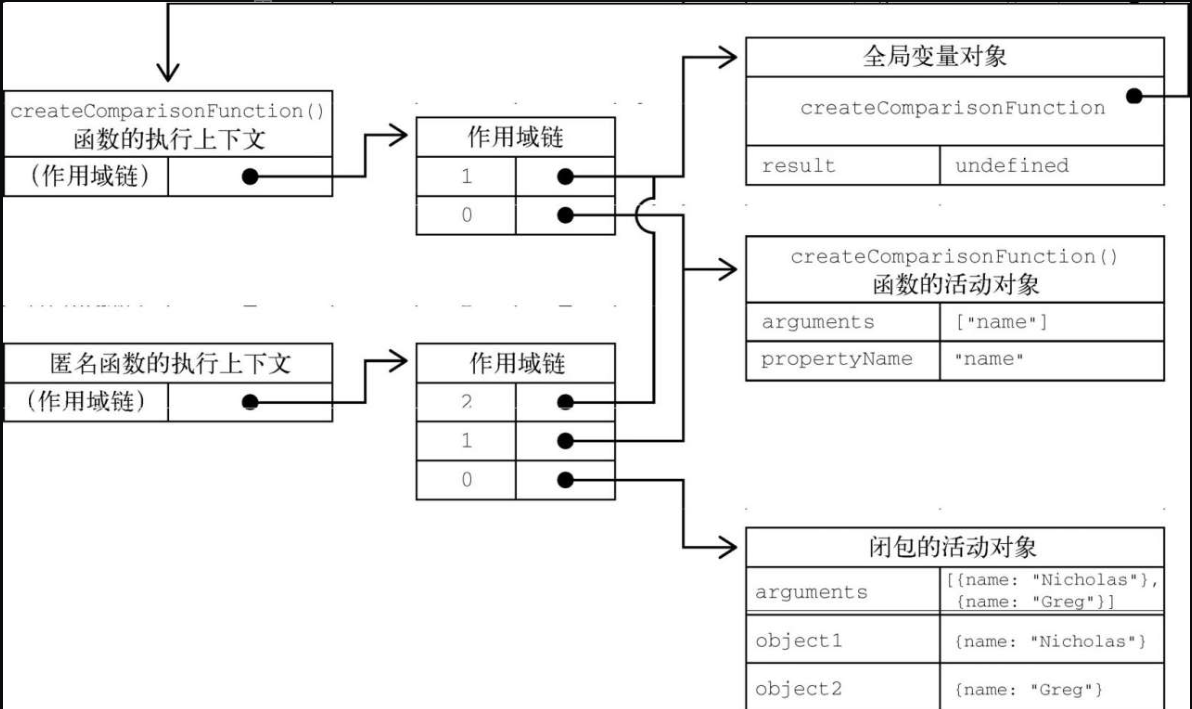
上面的代码中compare()函数的作用域链包含两个对象:

* 第一个对象是包含arguments、value1和value2的活动对象;
* 第二个对象是全局上下文的变量对象（包含this、result和compare）

如下图



闭包代码



// 创建比较函数

let compareNames = createComparisonFunction('name');

// 调用函数

let result = compareNames({ name: 'Nicholas' }, { name: 'Matt' });

// 解除对函数的引用，这样就可以释放内存了

compareNames = null;

**理解**

ES6中setTimeout函数的执行上下文

setTimeout函数中的this

[关于"this"的问题](https://developer.mozilla.org/zh-CN/docs/Web/API/setTimeout" \l "%E5%85%B3%E4%BA%8Ethis%E7%9A%84%E9%97%AE%E9%A2%98)

当你向 setTimeout() (或者其他函数) 传递一个函数时，该函数中的this指向跟你的期望可能不同，这个问题在 [JavaScript reference](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this" \l "method_binding) 中进行了详细解释。

**Chapter 10 - Functions**

Functions

WHAT’S IN THIS CHAPTER?

* Function expressions, **function declarations**, and arrow functions
* Default parameters and **spread operators**
* Recursion with functions
* Private variables using closures

**NO OVERLOADING**

ECMAScript functions cannot be overloaded in **the traditional sense**. In other languages, such as Java, it is possible to write two definitions of a function as long as their **signatures** (the type and number of arguments accepted) are different. As just discussed, functions in ECMAScript don’t have signatures.

because the arguments are represented as an array containing zero or more values. Without function signatures, true overloading is not possible.

If two functions are defined to have the same name in ECMAScript, it is the last function that

becomes the owner of that name. Consider the following example:

function addSomeNumber(num) {

return num + 100;

}

function addSomeNumber(num) {

return num + 200;

}

let result = addSomeNumber(100); // 300

Here, the function addSomeNumber() is defined twice. The first version of the function adds 100 to the argument, and the second adds 200. When the last line is called, it returns 300 because the second function has overwritten the first.

As mentioned previously, it’s possible to **simulate** overloading of methods by checking the type and number of arguments that have been passed into a function and then reacting accordingly.

Thinking of function names as pointers also explains why there can be no function overloading in

ECMAScript. In the previous example, it’s clear that declaring two functions with the same name

always results in the last function overwriting the previous one. This code is almost exactly equivalent to the following:

let addSomeNumber = function(num) {

return num + 100;

};

addSomeNumber = function(num) {

return num + 200;

};

let result = addSomeNumber(100); // 300

In this rewritten code, it’s much easier to see exactly what is going on. The variable addSomeNumber is simply being overwritten when the second function is created.

**SPREAD ARGUMENTS AND REST PARAMETERS**

ECMAScript 6 introduces the spread operator, which allows for a very **elegant** way of **managing**

**and grouping** collections. One of its most useful **applications** is in the **domain** of function signatures where it shines especially brightly **in the domain** of weak typing and variable length arguments.

The spread operator is **useful** both when invoking a function, as well as when defining a function’s parameters.

**Spread Arguments**

Instead of passing an array as a single function argument, it is often useful to be able to break apart an array of values and individually pass each value as a separate argument.

**Suppose** you have the following function defined, which sums all the values passed as arguments:

let values = [1, 2, 3, 4];

function getSum() {

let sum = 0;

for (let i = 0; i < arguments.length; ++i) {

sum += arguments[i];

}

return sum;

}

This function expects each of its arguments to be an individual number that will be iterated through to **find the sum**. An array outside the function containing all the values you want to sum is a logical format, but the most **prudent** way to **flatten** this array into separate parameters is to **inelegantly** utilize.apply():

console.log(getSum.apply(null, values)); // 10

In ECMAScript 6, you are now able to perform this action more **succinctly** using the spread operator. Applying the spread operator to an iterable object and passing that as a single argument to a function will **break apart** that iterable object of size N and pass it to the function as N separate arguments.

With the spread operator, you can **unpack the outer array** into **individual arguments** directly inside **the function invocation**:

console.log(getSum(...values)); // 10

Because the size of the array is known, there are no **restrictions** on other parameters **appearing** before or after the spread operator, including other spread operators:

console.log(getSum(-1, ...values)); // 9

console.log(getSum(...values, 5)); // 15

console.log(getSum(-1, ...values, 5)); // 14

console.log(getSum(...values, ...[5,6,7])); // 28

The **presence** of the spread operator is totally unknown **to the arguments objec**t; it will **treat** the

value being broken apart as separate pieces because that is how they are passed to the function:

let values = [1,2,3,4]

function countArguments() {

console.log(arguments.length);

}

countArguments(-1, ...values); // 5

countArguments(...values, 5); // 5

countArguments(-1, ...values, 5); // 6

countArguments(...values, ...[5,6,7]); // 7

The arguments object is **only** one way to **consume** spread arguments. Spread arguments can be used as named parameters in both **standard functions** and arrow functions, as well as **alongside** default arguments:

function getProduct(a, b, c = 1) {

return a \* b \* c;

}

let getSum = (a, b, c = 0) => {

return a + b + c;

}

console.log(getProduct(...[1,2])); // 2

console.log(getProduct(...[1,2,3])); // 6

console.log(getProduct(...[1,2,3,4])); // 6

console.log(getSum(...[0,1])); // 1

console.log(getSum(...[0,1,2])); // 3

console.log(getSum(...[0,1,2,3])); // 3

**Rest Parameter**

When **composing** a function definition, instead of handling parameters individually, it is possible to use the spread operator to **combine ranges of parameters** of variable length into a single array. In many ways, this is very similar to how the arguments object works, but in this case the rest parameter becomes a formal Array object.

function getSum(...values) {

// **Sequentially** sum all elements in 'values'

// Initial total = 0

return values.reduce((x, y) => x + y, 0);

}

console.log(getSum(1,2,3)); // 6

If there are named parameters **preceding** the rest parameter, it will **assume** the size of the remaining parameters that remain unnamed, or an empty array if there are none. Because the rest parameter is variable in size, you are only able to use it as the last formal parameter:

// Error

function getProduct(...values, lastValue) {}

// OK

function ignoreFirst(firstValue, ...values) {

console.log(values);

}

ignoreFirst(); // []

ignoreFirst(1); // []

ignoreFirst(1,2); // [2]

ignoreFirst(1,2,3); // [2, 3]

Although arrow functions do not support the arguments object, they do support rest parameters,

which **affords** you behavior that is extremely similar to arguments:

let getSum = (...values) => {

return values.reduce((x, y) => x + y, 0);

}

console.log(getSum(1,2,3)); // 6

As you might expect, using a rest parameter does not **affect** the arguments object—it will still exactly reflect what was passed to the function:

function getSum(...values) {

console.log(arguments.length); // 3

console.log(arguments); // [1, 2, 3]

console.log(values); // [1, 2, 3]

}

console.log(getSum(1,2,3))

**FUNCTION INTERNALS**

In ECMAScript 5, two special objects exist inside a function: arguments and this. In ECMAScript 6,

the new.target property was introduced.

**Arguments**.

The arguments object, as discussed previously, is an array-like object that contains all of the arguments that were passed into the function. It is only available when a function is declared using the function keyword (as **opposed to** arrow function declaration). Though its **primary use** is to **represent** function arguments, the arguments object also has a property named callee, which is a pointer to the function that owns the arguments object. **Consider** the following classic factorial function:

function factorial(num) {

if (num <= 1) {

return 1;

} else {

return num \* factorial(num - 1);

}

}

Factorial functions are typically defined to be recursive, as in this example, which works fine when the name of the function is set and won’t be changed. However, the proper execution of this function is tightly coupled with the function name "factorial". It can be **decoupled** by using arguments. callee as follows:

function factorial(num) {

if (num <= 1) {

return 1;

} else {

return num \* arguments.callee(num - 1);

}

}

In this rewritten version of the factorial() function, there is no longer a reference to the name

"factorial" in the function body, which ensures that the recursive call will happen on the correct

function no matter how the function is referenced. Consider the following:

let trueFactorial = factorial;

factorial = function() {

return 0;

};

console.log(trueFactorial(5)); // 120

console.log(factorial(5)); // 0

Here, the variable trueFactorial is assigned the value of factorial, effectively storing the function pointer in a second location. The factorial variable is then reassigned to a function that simply

returns 0. Without using arguments.callee in the original factorial() function’s body, the call to

trueFactorial() would return 0. However, with the function decoupled from the function name,

trueFactorial() correctly calculates the factorial, and factorial() is the only function that

returns 0.

**this**

The next special object is called this, which behaves differently when used inside a standard function and an arrow function.

Inside a standard function, it is a reference to the context object that the function is operating on—often called the this value (when a function is called in the global scope of a web page, the this object points to window). Consider the following:

window.color = 'red';

let o = {

color: 'blue'

};

function sayColor() {

console.log(this.color);

}

sayColor(); // 'red'

o.sayColor = sayColor;

o.sayColor(); // 'blue'

The function sayColor() is defined globally but references the this object. **The value of this is**

**not determined until the function is called**, so its value may not be consistent throughout the code execution. When sayColor() is called in the global scope, it outputs "red" because this is pointing to window, which means this.color evaluates to window.color. By assigning the function to the object o and then calling o.sayColor(), the this object points to o, so this.color evaluates to

o.color and "blue" is displayed.

Inside an arrow function, this references the context in which the arrow function expression is

defined. This is **demonstrated** in the following example, where two different invocations of sayColor both reference the property of the window object, which is the context inside which the arrow function was initially defined:

window.color = 'red';

let o = {

color: 'blue'

};

let sayColor = () => console.log(this.color);

sayColor(); // 'red'

o.sayColor = sayColor;

o.sayColor(); // 'red'

This is **especially** useful in situations where events or **timeouts** will invoke a function inside a callback where the invoking object is not the intended object. When an arrow function is used in these situations, the context referenced by this is preserved:

function King() {

this.royaltyName = 'Henry';

// 'this' will be the King instance

setTimeout(() => console.log(this.royaltyName), 1000);

}

function Queen() {

this.royaltyName = 'Elizabeth';

// 'this' will be the window object

setTimeout(function() { console.log(this.royaltyName); }, 1000);

}

new King(); // Henry

new Queen(); // undefined

NOTE Remember that function names are **simply** variables containing pointers,

so the global sayColor() function and o.sayColor() point to the same function

even though they execute in different contexts.

**caller**

ECMAScript 5 also **formalizes** an **additional** property **on a function object**: caller. Though not

defined in ECMAScript 3, all browsers except earlier versions of Opera supported this property,

which contains a reference to the function that called this function or null if the function was called from the global scope. For example:

function outer() {

inner();

}

function inner() {

console.log(inner.caller);

}

outer();

This code displays an alert with the source text of the outer() function. Because outer() calls

inner(), inner.caller points back to outer(). For looser coupling, you can also access the same

information via arguments.callee.caller:

function outer() {

inner();

}

function inner() {

console.log(arguments.callee.caller);

}

outer();

When function code executes in strict mode, attempting to access arguments.callee results

in an error. ECMAScript 5 also defines arguments.caller, which also results in an error in

strict mode and is always undefined outside of strict mode. This is to clear up confusion between arguments.caller and the caller property of functions. These changes were made as security additions to the language, so third-party code could not inspect other code running in the same context.

Strict mode places one additional restriction: you cannot assign a value to the caller property of a function. Doing so results in an error.

**new.target**

Functions have always been able to behave as both a constructor to instantiate a new object, and as a normal callable function. New in ECMAScript 6 is the ability to determine if a function was invoked with the new keyword using new.target. If a function is called normally, new.target will be undefined. If a function is called using the new keyword, new.target will reference the constructor or function.

function King() {

if (!new.target) {

throw 'King must be instantiated using "new"'

}

console.log('King instantiated using "new"';

}

new King(); // King instantiated using "new"

King(); // Error: King must be instantiated using "new"

**第11章 期约与异步函数**

异步逻辑的组织方式：Promise（期约）

异步函数的机制：async和await关键字

同步

执行顺序及其他异步行为的演示：

演示方式为异步日志输出setTimeout(console.log, 0, ...params)，好处是

浏览器控制台的输出经常能打印出JavaScript运行中无法获取的对象信息（比如期约的状态）

Promise：用来接收异步的结果，并在得到结果的时候进行某些操作

异步是等待某个结果返回，再进行某些处理操作

异步就是在等待某些长时间的操作或你不想为等待某个异步操作而阻塞线程执行时，异步是隔一段时间才能得到结果，异步是使用的手段。

1. 消息队列的概念
2. 同步与异步的概念：

同步即指指令严格按照出现的顺序执行，并且在执行时能立即获得存储在系统本地的信息

什么是异步操作：不让进程长时间等待或阻塞

1. 举例说明异步的概念，即为什么说这是一个异步
2. 同步与异步的执行过程
3. 回调模型是什么意思？回调地狱是怎么产生的？

11.2 期约

期约是对尚不存在结果的一个替身。期约（promise）这个名字最早是由Daniel

扩展

[ECMAScript® 2022 Language Specification (ecma-international.org)](https://262.ecma-international.org/13.0/" \l "sec-terms-and-definitions-prototype)

官网

Scope：适用范围

# 4.4.8 prototype

object that provides shared properties for other objects

注释：

SubType.prototype = new SuperType();

所谓的new一个构造函数，可以理解为给prototype对象添加属性和方法

NOTE

When a [constructor](https://262.ecma-international.org/13.0/" \l "constructor) creates an object, that object implicitly references the [constructor](https://262.ecma-international.org/13.0/" \l "constructor)'s "prototype" property for the purpose of resolving property references. The [constructor](https://262.ecma-international.org/13.0/" \l "constructor)'s "prototype" property can be referenced by the program expression **constructor.prototype**, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype. Alternatively, a new object may be created with an explicitly(显式地) specified prototype by using the **Object.create** built-in function.

# 20.2.4.3 prototype

Function instances that can be used as a [constructor](https://262.ecma-international.org/13.0/" \l "constructor) have a "prototype" property. Whenever such a Function instance is created another [ordinary object](https://262.ecma-international.org/13.0/" \l "ordinary-object) is also created and is the initial value of the function's "prototype" property. Unless otherwise specified, the value of the "prototype" property is used to initialize the [[Prototype]] internal slot of the object created when that function is invoked as a [constructor](https://262.ecma-international.org/13.0/" \l "constructor).

This property has the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }.

NOTE

[Function objects](https://262.ecma-international.org/13.0/" \l "function-object) created using **Function.prototype.bind**, or by evaluating a [MethodDefinition](https://262.ecma-international.org/13.0/" \l "prod-MethodDefinition) (that is not a [GeneratorMethod](https://262.ecma-international.org/13.0/" \l "prod-GeneratorMethod) or [AsyncGeneratorMethod](https://262.ecma-international.org/13.0/" \l "prod-AsyncGeneratorMethod)) or an [ArrowFunction](https://262.ecma-international.org/13.0/" \l "prod-ArrowFunction) do not have a "prototype" property.

# 27.3.4.3 prototype

Whenever a GeneratorFunction instance is created another [ordinary object](https://262.ecma-international.org/13.0/" \l "ordinary-object) is also created and is the initial value of the generator function's "prototype" property. The value of the prototype property is used to initialize the [[Prototype]] internal slot of a newly created Generator when the generator [function object](https://262.ecma-international.org/13.0/" \l "function-object) is invoked using [[Call]].

This property has the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }.

NOTE

Unlike Function instances, the object that is the value of the a GeneratorFunction's "prototype" property does not have a "constructor" property whose value is the GeneratorFunction instance.

# 27.4.4.3 prototype

Whenever an AsyncGeneratorFunction instance is created another [ordinary object](https://262.ecma-international.org/13.0/" \l "ordinary-object) is also created and is the initial value of the async generator function's "prototype" property. The value of the prototype property is used to initialize the [[Prototype]] internal slot of a newly created AsyncGenerator when the generator [function object](https://262.ecma-international.org/13.0/" \l "function-object) is invoked using [[Call]].

This property has the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }.

NOTE

Unlike function instances, the object that is the value of the an AsyncGeneratorFunction's "prototype" property does not have a "constructor" property whose value is the AsyncGeneratorFunction instance.

# 23.1.2.4 Array.prototype

The value of **Array.prototype** is the [Array prototype object](https://262.ecma-international.org/13.0/" \l "sec-properties-of-the-array-prototype-object).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

MATT FRISBIE-brief biography

李松峰小传

奇技指南

本文作者李松峰，资深技术图书译者，翻译出版过40余部技术及交互设计专著，现任360奇舞团Web[前端开发](https://so.csdn.net/so/search?q=%E5%89%8D%E7%AB%AF%E5%BC%80%E5%8F%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qihoo_tech/article/details/_blank)资深专家，360前端技术委员会委员、W3C AC代表

关于奇舞团

奇舞团是 360 集团最大的大前端团队，代表集团参与 W3C 和 ECMA 会员（TC39）工作。奇舞团非常重视人才培养，有工程师、讲师、翻译官、业务接口人、团队 Leader 等多种发展方向供员工选择，并辅以提供相应的技术力、专业力、通用力、领导力等培训课程。奇舞团以开放和求贤的心态欢迎各种优秀人才关注和加入奇舞团。

奇舞团十周岁生日快乐！

**原型研究**

是什么？

原型的用处是什么